Using MMX™ Instructions to Implement Data Alignment

Information for Developers and ISVs

From Intel® Developer Services
www.intel.com/IDS

March 1996

CONTENTS

1.0 INTRODUCTION

The media extension to the Intel Architecture (IA) instruction set includes single-instruction multiple-data (SIMD) instructions.This application note presents examples of code that use the MMX technology techniques that guarantee data alignment with Assembly, C, C++, or Microsoft Windows*.

When the fundamental data type was the byte, data alignment was not required for optimum microprocessor performance. Processors accessed bytes equally fast on even or odd boundaries. On later microprocessor architectures, 16-bit and larger .data must be aligned on even boundaries or extra time is required as the processor makes the mis-aligned data access.

With the introduction of the MMX™ instruction set for the Pentium® processor, data alignment is a valuable tool for developing premium performance applications because aligned data can be accessed per cycle while mis-aligned data accesses incur a three cycle penalty and mis-aligned accesses that span a cache line incur a twelve-plus cycle penalty.

For a simple demonstration of the value of data alignment, sum the elements of an array first on an aligned array, and then on a mis-aligned array. Once we have an aligned array, we can guarantee misalignment by adding one to the array index.

Example 1. Sum-aligned and Mis-aligned Data Buffers

```
;sum an aligned buffer
PADDD    MM0, buffer_A[00]
PADDD    MM0, buffer_A[08]
PADDD    MM0, buffer_A[16]
PADDD    MM0, buffer_A[24]
...
;sum a mis-aligned buffer
PADDD    MM0, buffer_B[01]        ;force mis-alignment
PADDD    MM0, buffer_B[09]
PADDD    MM0, buffer_B[17]
PADDD    MM0, buffer_B[25]
...
```

Table 1. Clock Cycles to Sum Data (Aligned and Mis-aligned)

Type	Number of Clock Cycles to Sum Array
Aligned data	76
Mis-aligned data	256

As shown in Table 1, significant performance gains (in excess of 300 percent) occur when summing aligned data.

To avoid the multiple access penalty for mis-aligned data on Pentium processors, the *Pentium® Processor Family Developer's Manual Volume 3: Architecture and Programming Manual* (Order Number 241430) lists the following data alignment rules:

- 2-byte data should be fully contained within an aligned 4-byte word
- 4-byte data should be on a 4-byte boundary
- 8-byte data should be aligned on an 8-byte boundary

Using MMX™ Instructions to Implement Data Alignment

March 1996

Data alignment is vital for developing high-performance applications with the two new data types for the MMX instruction set:

- Packed byte 8 bytes packed into one 64-bit data type
- Packed word Quadword packed into one 64-bit data type

For applications such as multimedia applications, whose fundamental data types consist of small data types, bytes and words, SIMD instructions process multiple data elements on single clock cycles. For instance, logical operations can be performed on eight 8-bit pixel elements on a single clock cycle. The new MMX instruction set provides a rich set of arithmetic and logical operations which operate on 64-bit registers as parallel registers of eight bytes, or four words, or two doublewords. If data is aligned and optimization provisions are met, instructions can access and process up to 16 bytes of data per clock cycle in the dual execution pipes of the Pentium processor.

2.0. DATA ALIGNMENT WITH STRUCTURES

2.1. Data Structures

Buffers are the fundamental data structure of multimedia programming. While it has not been the case until recently, tool developers are beginning to provide default array alignment based on the array data type. Arrays of words are aligned on word boundaries, and doublewords are aligned to 8-byte boundaries. Do not assume that arrays are aligned, but check the alignment behavior of your development tools.

2.1.1. Packed Data Structures

Because structures can consist of data items of various sizes and alignments, structures present a potential source of data mis-alignment. Programs compiled for Microsoft Windows* require structures to be packed on one-byte boundaries. Alignment of data items within structures can be controlled with either the /Zpn compiler option or the pack(n) pragma.

2.1.1.1. /Zpn

The /Zpn compiler option specifies the alignment of data within structures. Small data items are padded with extra bytes to the size specified by *n*. For Microsoft Visual C/C++* 1.5 *n* =1, 2, or 4. For Microsoft Visual C/C++ 4.0 n = 1, 2, 4, 8, 16.

2.1.1.2 #Pragma pack(n)

The pack(n) pragma aligns data in structures on *n* byte boundaries where *n* = 1, 2, 4, 8. The pack(n) pragma can be used to control packing on a per structure basis in source code.

Warning:

/Zpn and pack(n) are implementation specific and do not align structures on particular boundaries. They align data within structures.

2.2. Intel Reference Compiler

The default behavior of the Intel Reference Compiler is to set 8 bytes as the strictest alignment constraint. Structures are aligned according to the largest alignment requirement. If the largest component is a double, the smaller elements of a data structure will be padded to double size. The padding behavior can be overridden with pack(n).

3.0. DATA ALIGNMENT IN ASSEMBLY

3.1. SEGMENT Directive

By default, Microsoft Macro-Assembler* (MASM) aligns segments on 16-byte paragraph addresses. The example below defines two 100 element quadword, 8-byte, type buffers, EXAMPLETbl and TESTTbl. Each is aligned to 16-byte boundaries because each is defined in its own segment. The ES segment register is assigned the segment address of the TESTTbl buffer.

Example 2. Alignment with the SEGMENT Directive

```
EXAMPLESEG SEGMENT PARA USE16 PUBLIC 'DATA'
EXAMPLETbl       DQ 100 DUP (0AAAAAAAAAAAAAAAAh)
EXAMPLESEG ENDS
TESTSEG SEGMENT PARA USE16 PUBLIC 'DATA'
TESTTbl          DQ 100 DUP (0555555555555555h)
TEST_SEG ENDS
        ;
        ;
        ;
EXAMPLECODE SEGMENT USE16 PUBLIC 'CODE'
        ASSUME CS:EXAMPLECODE, DS:EXAMPLESEG, ES:TESTSEG, SS:EXAMPLESTACK
Begin:
        mov             ax, EXAMPLESEG
        mov             ds, ax
        mov             ax, TESTSEG
        mov             es, ax
        ;
        ;
        ;
EXAMPLECODE ENDS
        end Begin
```

3.2. ALIGN Directive

The ALIGN *n* directive aligns data on arbitrary *n* boundaries. In the example below, data bytes are defined to force mis-alignment of the EXAMPLETbl. The ALIGN directive aligns EXAMPLETbl on the next 8-byte boundary.

Example 3. Alignment with the Align Directive

```
;Alignment with the ALIGN n directive
EXAMPLEDATA SEGMENT PARA USE16 PUBLIC 'DATA'
        DB 3 DUP (0)     ;As a test, force mis-alignment
ALIGN 8
EXAMPLETbl       DQ 100 DUP (0AAAAAAAAAAAAAAAAh)
EXAMPLEDATA ENDS
        ;
        ;
        ;
EXAMPLECODE SEGMENT PARA PUBLIC 'CODE'
        ASSUME CS:EXAMPLECODE, DS:EXAMPLEDATA, SS:EXAMPLESTACK
Begin:
        ;
        ;
        ;
EXAMPLECODE ENDS
```

```
        end Begin
```

3.3. Data Pointers

This method involves adjusting a pointer with an offset to create a pointer to an arbitrarily aligned block of data. The pointer Pointed to is decrement by the number of bytes required to create a pointer the at points to an *n* to aligned block of data. An extra data element has been defined before the buffer definition the provide offset memory.

Example 4. Alignment with an Indirect Pointer

```
; Demonstrates data alignment with an indirect pointer
EXAMPLEDATA SEGMENT PARA USE16 PUBLIC 'DATA'
        DQ  0    0AAAAAAAAAAAAAAAAh
EXAMPLETbl      DQ 100 DUP (0AAAAAAAAAAAAAAAAh)
EXAMPLETblBase dw 0
EXAMPLEDATA ENDS
        ;
        ;
        ;
EXAMPLECODE SEGMENT PARA USE16 PUBLIC 'CODE'
        ASSUME CS:EXAMPLECODE, DS:EXAMPLEDATA, SS:EXAMPLESTACK
Begin:
        ;
        ;
        ;
        mov             ax, OFFSET EXAMPLETbl
        mov             bx, ax
        and             bx, 07h
        sub             ax, bx
        mov             EXAMPLETblBase, ax
        ;
        ;
        ;
EXAMPLECODE ENDS
        end Begin
EXAMPLETblBase is then used for data accesses.
```

3.4. LOCAL Directive Allocated Variables on the Stack

When the LOCAL directive is used it must be located immediately after the PROC directive. There is no way to force alignment before the LOCAL directive allocates space for locals. It would be convenient if there was a form of the ALIGN directive that could be used with LOCAL to align locals. Alignment can be achieved by adjusting a pointer as demonstrated above.

Example 5. Alignment of Local Variables with an Indirect Pointer

```
; Demonstrates data alignment of LOCAL directive stack variables
        ;
        ;
        call TASK
        ;
        ;
TASK    PROC
        LOCAL   nuts[8]:BYTE
        LOCAL           locBuff[10]:QWORD
        LOCAL           locPtr:WORD
```

```
        lea             ax, locBuff
        and             ax, 0FFF8h
        lea             bx, locPtr
        mov             word ptr ss:[bx], ax
        ret
TASK    ENDP
```

3.5. Adjusting STACK alignment with EBP or ESP

EBP and ESP can be aligned to reference arbitrary aligned regions on the stack.

Example 6. Aligning EBP

```
; Demonstrates aligning EBP to an aligned region on the stack

        push            ebp
        mov             eax, esp
        and             eax, 7
        sub             ebp, eax
        ...
        ;assembly routine code

        pop             ebp
        mov             esp, ebp
        pop             ebp
        ret
```

If ESP/SP is to be alignment adjusted, either the original ESP/SP or the adjustment value must be saved in order to return the stack pointer to its original alignment. The example below saves the stack pointer.

Example 7. Aligning ESP

```
; Demonstrates aligning EBP to an aligned region on the stack
;entry code
        ...
mov             eax, esp
mov             savedStackPointer, eax          ;save copy of mis-aligned stack pointer
and             esp, 0FFFFFFF8h                 ;align stack pointer
...                                             ;program code
mov             eax, savedStackPointer
mov             esp, eax                        ;Restore stack pointer
;exit code
...
ret
```

4.0. ALIGNMENT IN C, C++ and WINDOWS*

The indirect method of data alignment is a flexible method of alignment. It is applicable to all levels of code development from Assembly to C++. It entails allocating a buffer of the required size plus an extra data element of the same size as the buffer elements. An offset is added to a pointer into the buffer that is to be data aligned. The pointer is then used for all subsequent data accesses. If data is to be freed the original pointer returned by the memory allocation function should be used to free memory.

4.1. Alignment for C with Malloc

Alignment in C code is similar to the indirect method in Assembly. A calculation is made to determine the offset in bytes required to align a pointer on an arbitrary boundary. The original pointer should be saved to free memory.

Example 8. Alignment with Malloc

```
int                     size_of_array = 100;
int                     iSizeOfDouble;
double *                ptrDoubleMem;
size_t          size;
void *          vptrMalloc;
unsigned int    iAdjust;

// size of buffer data element
iSizeOfDouble = sizeof(double);
// number of bytes to allocate
size = size_of_array * iSizeOfDouble + iSizeOfDouble;
// allocate memory
vptrMalloc = malloc(size);
// determine adjustment required to align pointer
iAdjust = (unsigned int)vptrMalloc % (unsigned int) iSizeOfDouble;
// calculate new aligned pointer
ptrDoubleMem = (double *)((unsigned int)(iSizeOfDouble-iAdjust)
       +(unsigned int)vptrMalloc);
// Program code
// free memory with the original pointer
free(vptrMalloc);
```

4.2. Alignment for Windows* with GlobalAlloc

Alignment in Windows is similar to the method above for C. For Windows memory allocation use GlobalAlloc instead of malloc from the standard C library. The pointer returned by GlobalAlloc is passed to GlobalLock to lock the allocated memory and prevent windows from reusing the memory. The pointer returned by GlobalLock is the pointer used for alignment adjustment.

Example 9. Alignment in Windows* with GlobalAlloc

```
int                     size_of_array = 100;
int                     iSizeOfDouble;
double FAR *    ptrDoubleMem;
DWORD           size;
HGLOBAL         hGlobalMemory;
void FAR *      lpGlobalMemory;
unsigned int    iAdjust;
```

```
// size of buffer data element
iSizeOfDouble = sizeof(double);

// number of bytes to allocate
size = size_of_array * iSizeOfDouble + iSizeOfDouble;

// allocate and lock memory
hGlobalMemory = GlobalAlloc(GMEM_MOVEABLE, size);
lpGlobalMemory = GlobalLock (hGlobalMemory);

// determine adjustment required to align pointer
iAdjust = (unsigned long)lpGlobalMemory % (unsigned long) iSizeOfDouble;
// calculate new aligned pointer
ptrDoubleItem = (double *)((unsigned long)( iSizeOfDouble - iAdjust) +
       (unsigned long)lpGlobalMemory);
// Program code
// free memory with the original pointer
GlobalUnlock(hGlobalMemory);
GlobalFree(hGlobalMemory);
```

4.3. Alignment for C++ with NEW

In C++, use NEW to allocate memory. The original pointer should be saved in order to free memory with delete.

Example 10. Alignment In C++ with NEW

```
int                 size_of_array = 100;
char *         ptrNewMem;
double *            ptrDoubleMem;
int                 i;
size_t         iSizeOfDouble;
unsigned int   iAdjust;
unsigned long size;
// size of buffer data element
iSizeOfDouble = sizeof(double);
// number of bytes to allocate
size = size_of_array * iSizeOfDouble + iSizeOfDouble;
// allocate memory
ptrNewMem = new char[size];
// determine adjustment required to align pointer
iAdjust = (unsigned int)ptrNewMem % (unsigned int) iSizeOfDouble;
// calculate new aligned pointer
ptrDoubleMem = (double *)((unsigned int)( iSizeOfDouble -iAdjust)
       +(unsigned int)ptrNewMem);
// Program code
// free memory with the original pointer
delete [] ptrNewMem;
```